Extending Task Parallelism For Frequent Pattern Mining

Prabhanjan Kambadur [a], Amol Ghoting [b], Anshul Gupta [b] and
Andrew Lumsdaine [a]

[a] *Open Systems Lab, Indiana University, Bloomington, IN - 47408*
[b] *IBM T J Watson Research Center, Yorktown Heights, NY - 10598*

Abstract. Algorithms for frequent pattern mining, a popular informatics application, have unique requirements that are not met by any of the existing parallel tools. In particular, such applications operate on extremely large data sets and have irregular memory access patterns. For efficient parallelization of such applications, it is necessary to support dynamic load balancing along with scheduling mechanisms that allow users to exploit data locality. Given these requirements, task parallelism is the most promising of the available parallel programming models. However, existing solutions for task parallelism schedule tasks implicitly and hence, custom scheduling policies that can exploit data locality cannot be easily employed. In this paper we demonstrate and characterize the speedup obtained in a frequent pattern mining application using a custom clustered scheduling policy in place of the popular Cilk-style policy. We present PFunc, a novel task parallel library whose customizable task scheduling and task priorities facilitated the implementation of our clustered scheduling policy.

Keywords. task parallelism, frequent pattern mining, data locality.

Introduction

Algorithms for frequent pattern mining (FPM), a popular informatics application, exhibit certain distinct characteristics that distinguish them from traditional high-performance computing applications. Typically, FPM applications operate on large, irregular and dynamic data sets. The data (and corresponding computations) cannot be partitioned *a priori*, making these applications extremely sensitive to load balancing and scheduling. Memory access patterns are often data-dependent, requiring one data object's location in memory to be resolved before the next can be fetched. Finally, safe parallelization of FPM applications requires fine-grained synchronization. For all of these reasons, popular parallel programming models such as the data parallel and the single process multiple data (SPMD) models are not well-suited to FPM applications. Of the variety of alternative parallel programming models available, task parallelism is the most promising when it comes to meeting the challenges posed by FPM applications. The task parallel programming model is sufficiently high-level and general pur-

pose to be able to parallelize both regular and irregular applications. However, existing solutions for task parallelism have shortcomings that prevent efficient parallelization of FPM applications. Specifically, FPM applications require custom task scheduling policies that can exploit data locality between tasks that are not always related by the parent-child relationship. In most existing solutions for task parallelism, not only are tasks scheduled transparently from the users, but also, data locality between tasks is exploited only if there is a parent-child relationship between those two tasks.

In this paper, we demonstrate the speedup obtained in a task parallel Apriori-based FPM implementation by switching from the popular Cilk-style [8] task scheduling policy to a customized "clustered" task scheduling policy. Furthermore, we collect various hardware metrics to characterize the factors that resulted in the speedup. To implement these two (Cilk-style and clustered) scheduling policies, we use PFunc, a novel library-based implementation of task parallelism, that allows users to customize parameters such as task scheduling policy and task priority. Unlike most other parallelization tools, PFunc provides a natural interface that enables facile implementation of our clustered scheduling policy. Through this study, we highlight the need to incorporate support for efficient parallelization of FPM applications in the task parallel model.

1. Background

Murphy and Kogge [23] demonstrated the differences in memory access patterns of informatics applications when compared to those of traditional scientific computing applications. Berry et al. [7] and Lumsdaine et al. [22] have shown that current techniques applied to high performance computing are inadequate for informatics applications. Since FPM was introduced as a relevant problem in informatics, its memory characteristics and parallelization have been extensively researched [3, 4, 9, 16, 18, 19, 21, 24, 26, 29]. Zaki and Parathasarathy [31] were the first to explore the idea of clustering to aid in fast discovery of frequent patterns.

Many solutions have been implemented to facilitate dynamic task parallelism. Fortran M [13], Cilk [14] and OpenMP 3.0 [25] implement task parallelism as extensions to stock programming languages. Other solutions such as Intel's Threading Building Blocks (TBB) [27], Microsoft's Parallel Patterns Library (PPL) and Task Parallel Library (TPL), and Java Concurrency Utilities are library-based. All of the three HPCS languages (Chapel [10], Fortress [6] and X10 [11]) offer task parallelism as language features. Scheduling of tasks has received wide attention in the programming community. Cilk's *depth-first work* [8] model, the X10 Work Stealing framework's (XWS) *breadth-first* [12] model and Guo et al.'s *hybrid* model [17] are notable examples of work stealing schedulers. Each of the three above mentioned scheduling policies exploit data locality under different circumstances. For example, Cilk's scheduling policy exploits data locality only when the applications are deeply nested. All of the task parallel solutions discussed above schedule tasks implicitly, thereby disallowing customization of task scheduling policies. In this paper, we demonstrate the importance of customizing the task scheduling policy in parallelization tools when

parallelizing FPM applications. We also describe the features of PFunc that enable FPM applications to employ custom scheduling strategies that help them outperform other implementations that employ default scheduling policies provided by existing solutions for task parallelism.

2. Problem Description

In this section, we describe FPM and comment on important aspects of its implementation that influence performance. Briefly, the problem description is as follows: Let $I = \{i_1, i_2, .., i_n\}$ be a set of n items, and let $D = \{T_1, T_2, .., T_m\}$ be a set of m transactions, where each transaction T_i is a subset of I. An itemset $i \subseteq I$ of size k is known as a k-itemset. The support (that is, frequency) of i is $\sum_{j=1}^{m}(1 : i \subseteq T_j)$, or informally speaking, the number of transactions in D that have i as a subset. The FPM problem is to find all $i \in D$ that have support greater than a user supplied minimum value. For our FPM implementation, we choose the *Apriori* [5] algorithm. Due to its efficiency, robustness and guaranteed main memory footprint, the Apriori algorithm is widely used in FPM implementations including those in commercial products such as IBM's InfoSphere Warehouse [30]. Apriori traverses the itemset search space in breadth-first order. Its efficiency stems from its use of the anti-monotone property: If a size k-itemset is not frequent, then any size *(k+1)*-itemset containing it will not be frequent. The algorithm first finds all frequent *1*-items in the data set, and then iteratively finds all frequent *k*-itemsets using the frequent *(k-1)*-itemsets discovered previously. For example, let A, B, C and D be individual items (*1*-itemsets) that are frequent in a transaction database. Then, for stage 2, AB, AC, AD, BC, BD, and CD are the candidate *2*-itemsets. If at stage 2, after counting, *2*-itemsets AB, AC and AD were found to be frequent, then ABC, ABD, and ACD are the candidates for stage 3. The frequency of a candidate *k*-itemset is counted by performing a join ($\bowtie$) operation on the transaction-ID lists of each individual item in that particular itemset. In our task parallel implementation of the Apriori algorithm, the counting operations required for each *k*-itemset are executed as a separate task.

Requirements for efficient parallelization: The Apriori algorithm is highly dependent on memory reuse for its performance [16]. As tasks mine for itemsets, they access overlapping memory regions as many itemsets share transaction-ID lists. For example, if we were mining for *2*-itemsets AB, AC, and AD, then the transaction-ID list of A would be common in the mining operations. The greater the overlap of items in the itemsets, the greater the potential for memory reuse between their respective tasks. Exploiting such inter-task data localities through locality-aware scheduling of tasks is the key for efficient parallel execution of Apriori-based FPM applications.

Shortcomings of current solutions: Existing task parallel solutions schedule tasks implicitly using different flavors of the Cilk-style work stealing [8] task scheduler and hence, do not support custom task scheduling policies that can exploit data localities between user-specified tasks. Furthermore, in the Cilk-style scheduling policy, there is little or no contention when a thread executes tasks

that are on its own task queue, but stealing a task from another thread's task queue is an expensive operation. Cilk-style work stealing benefits applications that are deeply nested (that is, recursive) in nature. Nested tasks, by definition, spawn other tasks. Therefore, when a thread steals a nested task, it implicitly gains all the tasks that will be generated by the stolen task–thereby minimizing task stealing. However, in the Apriori algorithm, tasks are non-nested as we traverse the search space in breadth-first order. Hence, once a thread runs out of work, it must steal tasks repeatedly from other *victim* threads' task queues in order to keep itself busy. Such work stealing is detrimental to Apriori-based FPM implementations' performance both because of the increased contention on victim threads' task queues and the lack of data locality among stolen tasks.

3. PFunc: A new tool for task parallelism

PFunc [20] is a lightweight and portable task parallel library for C and C++ users, which has been designed using the generic programming paradigm [15] to overcome some of the shortcomings of existing solutions for task parallelism. Due to space constraints, we describe only those features of PFunc that affect Apriori-based FPM implementations. PFunc allows users to choose the task scheduling policy at compile time. Users can either choose from built-in scheduling policies such as Cilk-style, first-in first-out (FIFO), last-in first-out (LIFO) and priority-based, or they can supply their own scheduling policy. All scheduling policies are "models" of the scheduler "concept". This generic design enforces a uniform interface across the different scheduling policies and enables compile time plug-and-play capabilities with no runtime penalty. By default, PFunc follows the work stealing model in which each thread has its own task queue and tasks are enqueued on the task queue of the thread that spawned the task. Users have the option to override this default setting at runtime and place tasks onto a particular thread's task queue (that is, change the task's *affinity*). PFunc also provides *task attributes* that can be attached to each individual task when spawning the task. Task attributes, such as task priority, are an important tool in the implementation of many scheduling policies. For example, when using priority-based scheduling, users specify a task's priority using the task attribute mechanism. Like the scheduler, task attributes can be customized to suit the task scheduling policy. To enable hardware profiling, PFunc is fully integrated with the Performance Application Programming Interface (PAPI) [1].

4. Clustered task scheduling

To efficiently parallelize our Apriori-based FPM implementation using the task parallel programming model, we have designed a clustered scheduling policy. In this policy, tasks are clustered together such that there is a greater likelihood of memory reuse between the clustered tasks (for example, tasks mining for *3*-itemsets ABC and ABD) as they are executed by the same thread. Furthermore, we employ a custom task stealing policy in which clusters of tasks are stolen

instead of a single task. Such clustered stealing not only reduces contention on the thread-local task queues by minimizing the number of required steals, but also maintains data locality between the stolen tasks.

Implementation: In our FPM implementation, each task performs the mining operations required for one *k*-itemset. Each *k*-itemset is stored as a *sorted* set of individual items. To cluster tasks that have significant memory overlap, we use a common prefix of length (k-1). For example, the *3*-itemsets ABC and ABD share the 2-prefix AB and hence, their respective tasks are clustered together. For efficient execution of our FPM application, it is necessary to ensure that such clusters of tasks have a high likelihood of being executed by the same thread. To this end, we use a hash table (std :: hash_map) as the task queue for each thread. This is a departure from the Cilk model [14], in which each thread's task queue is a deque. With the hash table structure in place, clustering of tasks is achieved by placing all *k*-itemsets that have a common (k-1) prefix into the same bucket. To achieve such clustering, we used the following hash function to compute the hash of a *k*-itemset. First, the hash of each of its first (k-1) items is computed using C++ Standard Library's [28] std :: hash function object. Then, these individual hashes are XOR'ed together to produce one final hash. For example, the hash for the *3*-itemset ABC is computed by XOR-ing the results of applying std :: hash to A and B separately. The hashes thus computed for the *3*-itemsets ABC and ABD are equal and hence, their respective tasks are placed in the same bucket. Each thread executes the tasks placed in its task queue (that is, hash table) by iterating through its task queue's buckets starting from the first non-empty bucket. When a thread runs out of tasks on its own task queue, it randomly selects a victim thread to steal tasks. Then, it steals the first non-empty bucket from the victim thread's task queue. This stealing policy is better equipped than the Cilk-style stealing policy to avoid repeated stealing in our FPM implementation as potentially, more than one task can be stolen during each steal. Furthermore, by stealing an entire bucket of tasks, data locality between stolen tasks is preserved.

Integrating with PFunc: Realizing our customized clustered task scheduling policy in PFunc is achieved in two steps. First, the entire scheduling policy is implemented to model PFunc's scheduler concept. This enables us to pick our clustered scheduling policy as the task scheduling policy of choice at compile time. Second, we customize PFunc's task attributes (again, at compile time) such that we can attach a *reference* to the *k*-itemset that needs to be mined as the respective task's priority. When a task is spawned, our hash function uses its priority (that is, the associated *k*-itemset's reference) to determine the appropriate bucket (that is, cluster) for this task in the spawning thread's task queue.

5. Results

In this section, we present the results of running our Apriori-based FPM implementation using the Cilk-style and the clustered scheduling policies. PFunc was used for parallelization of our FPM implementation and the scheduling policy for a particular run was chosen at compile time. Minimal code modifica-
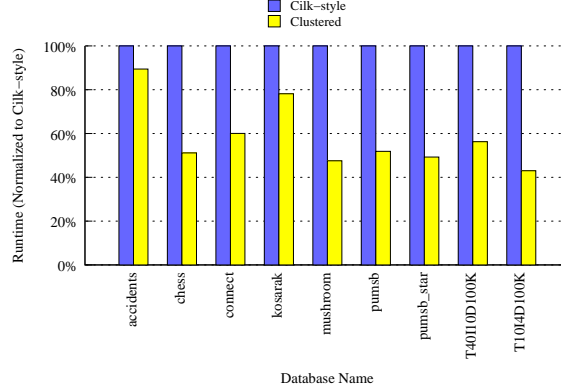
Figure 1. Graph showing the normalized runtimes of different datasets when using the Cilk-style and Clustered task scheduling policies in our Apriori-based FPM implementation with 8 threads. Support (that is, frequency) for each of the datasets is given in Table 1.

Dataset	Support	IPC		DTLB L1M/L2H		DTLB L1M/L2M	
		Cilk	Cluster	Cilk	Cluster	Cilk	Cluster
accidents	0.25	0.595689	0.603959	0.000048	0.000046	0.000161	0.000110
chess	0.6	0.560538	0.668965	0.000797	0.000242	0.001006	0.000032
connect	0.8	0.543099	0.809308	0.000249	0.000112	0.001204	0.000141
kosarak	0.0013	0.692103	0.717599	0.000400	0.000185	0.000659	0.000123
pumsb	0.75	0.494539	0.719072	0.000230	0.000114	0.001276	0.000126
pumsb_star	0.3	0.527659	0.698358	0.000315	0.000145	0.001082	0.000113
mushroom	0.10	0.570390	0.705003	0.000477	0.000267	0.000950	0.000022
T40I10D100K	0.005	0.627272	0.727288	0.000368	0.000305	0.000900	0.000021
T10I4D100K	0.00006	0.555330	0.716282	0.000218	0.000144	0.000876	0.000044

Table 1. Instructions-per-cycle (IPC), L1 data TLB misses (L1M/L2H) and L2 data TLB misses (L1M/L2M) when using the Cilk-style and Clustered task scheduling policies in our Apriori-based FPM implementation with 8 threads for different datasets from the FIMI repository [2].

tion was required to attach the *k*-itemset reference as task priority when spawning the tasks for the clustered scheduling policy. We ran our experiments on a four socket, quad-core (total 16 cores) AMD 8356 processor running Linux Kernel 2.6.24. For compilation, we used GCC 4.3.2 with: "-O3 -fomit-frame-pointer -funroll-loops". To collect hardware metrics, we used PFunc's integration with PAPI. The benchmarks were run on data sets from the FIMI repository [2].

Figure 1 depicts the runtimes of our FPM implementation for both the Cilk-style and the clustered task scheduling policies. Runtimes were recorded with hardware profiling turned off and were averaged after 5 runs. The clustered scheduling policy runs significantly faster (more than 50%) for most of the data sets, with the accidents.dat data set being the only exception. To test our hypothesis that our clustered scheduling policy exploits data locality better than the Cilk-style policy, we collected various hardware metrics. Table 1 summarizes some of the important metrics. Our clustered scheduling policy delivers more

instructions per clock cycle than the Cilk-style scheduling policy for all data sets. Also, our clustered scheduling policy incurs far fewer L2 data TLB misses than the Cilk-style scheduling policy. This is because the benefits of clustering tasks that have a significant memory overlap outweigh the additional operational costs incurred due to using hash tables in our clustered scheduling policy. When results from Figure 1 and Table 1 are taken together, we can conclude that the clustered scheduling policy exploits data locality better than the Cilk-style scheduling policy for our Apriori-based FPM implementation.

6. Conclusion and Future Work

We have demonstrated that our custom clustered scheduling policy performs better than the popular Cilk-style scheduling policy for an Apriori-based FPM implementation. This was made possible by PFunc, which provides better support for parallelization of FPM applications than existing solutions for task parallelism by allowing facile customization of task scheduling policy and task attributes. An interesting topic for future research is to implement a dynamic task scheduling policy that utilizes a multi-dimensional index structure as task queues. This would enable a thread to dynamically pick the "nearest-neighbor" of the previously executed task as its next task to execute.

References

[1] *Performance Application Programming Interface*. ICL, Knoxville, TN.

[2] *Datasets of the workshops on Frequent Itemset Mining Implementations (FIMI)*. University of Helsinki, 2004.

[3] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 207–216, New York, NY, USA, 1993. ACM.

[4] R. Agrawal and J. C. Shafer. Parallel Mining of Association Rules. *IEEE Trans. on Knowl. and Data Eng.*, 8(6):962–969, 1996.

[5] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules in Large Databases. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[6] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. The Fortress Language Specification, Version 1.0. Technical report, Sun Microsystems, Inc., 2008.

[7] J. W. Berry, B. Hendrickson, S. Kahan, and P. Konecny. Software and Algorithms for Graph Queries on Multithreaded Architectures. *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, 2007.

[8] R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS*, pages 356–368, 1994.

[9] G. Buehrer, S. Parthasarathy, and Y.-K. Chen. Adaptive Parallel Graph Mining for CMP Architectures. *Data Mining, IEEE International Conference on*, 0:97–106, 2006.

[10] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, Jan 2007.

[11] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[12] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen. Solving Large, Irregular Graph Problems Using Adaptive Work-Stealing. In *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, pages 536–545, Washington, DC, USA, 2008. IEEE Computer Society.

[13] I. Foster, D. R. Kohr, Jr., R. Krishnaiyer, and A. Choudhary. A Library-based Approach to Task Parallelism in a Data-parallel Language. *J. Parallel Distrib. Comput.*, 45(2):148–158, 1997.

[14] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, June 1998. Proceedings published in ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[15] R. Garcia, J. Järvi, A. Lumsdaine, J. Siek, and J. Willcock. An extended comparative study of language support for generic programming. *Journal of Functional Programming*, 2005.

[16] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y.-K. Chen, and P. Dubey. Cache-conscious frequent pattern mining on modern and emerging processors. *The VLDB Journal*, 16(1):77–96, 2007.

[17] Y. Guo, R. Barik, R. Raman, and V. Sarkar. Work-First and Help-First Scheduling Policies for Async-Finish Task Parallelism. In *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, May 2009.

[18] E.-H. S. Han, G. Karypis, and V. Kumar. Scalable Parallel Data Mining for Association Rules. *IEEE Trans. on Knowl. and Data Eng.*, 12(3):337–352, 2000.

[19] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 1–12, New York, NY, USA, 2000. ACM.

[20] P. Kambadur, A. Gupta, A. Ghoting, H. Avron, and A. Lumsdaine. PFunc: Modern Task Parallelism For Modern High Performance Computing. In *SC '09: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, Portland, Oregon, November 2009.

[21] J.-S. Kim, X. Qin, and Y. Hsu. Memory characterization of a parallel data mining workload. In *WWC '98: Proceedings of the Workload Characterization: Methodology and Case Studies*, page 60, Washington, DC, USA, 1998. IEEE Computer Society.

[22] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. W. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(1):5–20, 2007.

[23] R. C. Murphy and P. M. Kogge. On the Memory Access Patterns of Supercomputer Applications: Benchmark Selection and Its Implications. *IEEE Trans. Comput.*, 56(7):937–945, 2007.

[24] S. Nijssen and J. N. Kok. A quickstart in frequent structure mining can make a difference. In *KDD '04: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 647–652, New York, NY, USA, 2004. ACM.

[25] OpenMP Architecture Review Board. *OpenMP Application Program Interface, v3.0*. May 2008.

[26] S. Parthasarathy, M. J. Zaki, M. Ogihara, and W. Li. Parallel data mining for association rules on shared memory systems. *Knowl. Inf. Syst.*, 3(1):1–29, 2001.

[27] J. Reinders. *Intel Threading Building Blocks*. O'Reilly, 2007.

[28] A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project, May 1994.

[29] C. Wang and S. Parthasarathy. Parallel algorithms for mining frequent structural motifs in scientific data. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 31–40, New York, NY, USA, 2004. ACM.

[30] M. J. Zaki. Parallel and Distributed Association Mining: A Survey. *IEEE Concurrency*, 7(4):14–25, 1999.

[31] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New Algorithms for Fast Discovery of Association Rules. Technical report, Rochester, NY, USA, 1997.